19｜日志隔离：如何落地日志隔离？

2021-11-30 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼

时长 16:16 大小 14.91M

你好，我是高楼。

这节课，我们详细来讲讲如何基于微服务技术落地日志隔离。

在全链路压测的隔离中，有一个很重要的部分就是日志隔离，即区分对应的日志。之所以要将正常日志和压测日志都进行隔离，是为了不影响正常日志存储和方便线上问题排查。

在当前的技术市场上，主流的日志框架是 Log4j、Log4j2（Log4j 的升级版）和 Logback。

Logback 是 Log4j 的改良版，拥有更多的特性，性能也更好，天然支持 SLF4J（ Simple Logging Facade for Java，日志输出门面，非日志框架）。Slf4j 可以支持不同的日志框

架，实现对日志框架的一层抽象，可以整理不同的日志框架的输出。我们这个电商项目也是使用的 Logback 的方式。

在具体的方案选择上，我们选择改写 Logback 的 Appender，然后根据压测标记将日志写入对应目录中。

既然已经选定了 Logback 日志框架做日志隔离改造，那么我们就可以直接上手进行技术预演了。

技术预演

首先我们还是先在本地开发环境做一下日志隔离的技术预演。预演通过之后，再把相关代码移植到 demo 系统中去验证。

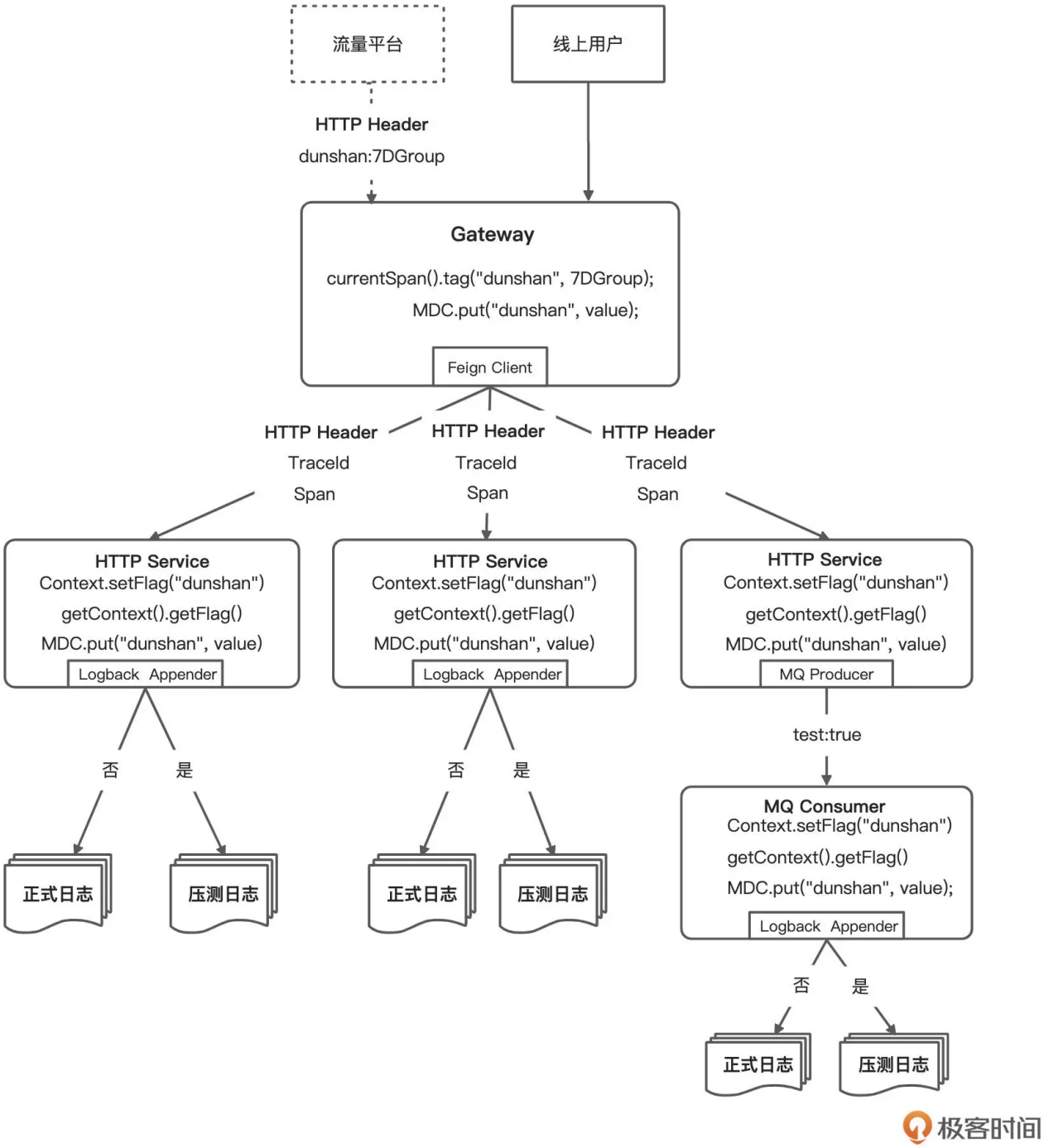
在第 14-18 讲，我们已经通过 Sleuth 把压测标记透传到了每个服务中，下面只需要从对应服务的数据上下文获取标记，然后完成逻辑判断就能实现日志的分离了。

考虑到日志打印一般是无序的，在一个用户的使用过程中，可能有多个不同的线程来进行处理。典型的例子是当用户访问某个页面时，应用服务可能会创建一个新的线程来处理该请求，也可能从线程池中复用已有的线程。在一个用户 session 存续期间，可能有多个线程处理过该用户的请求。当需要追踪某个用户在系统中的相关日志记录时，就会变得很麻烦。

解决办法就是引入 MDC（Mapped Diagnostic Context，映射调试上下文）。而 SLF4J 提供了 MDC（Mapped Diagnostic Contexts）功能，它的实现也是利用了 ThreadLocal 机制。改造代码只需要将指定的值 put 到线程上下文的 Map 中，然后在对应的地方使用 get 方法获取对应的值，从而达到自定义和修改日志输出格式内容的目的。

考虑到压测流量比较大，产生的日志也非常多，压测流量的日志很容易覆盖正常流量的日志。我们选择改写 Logback 的 Appender，根据流量标记将日志写入对应的影子目录中，并且配置默认的删除策略，日志只保存三天。

整体方案设计可以参考下面这张图：



下面我们来看一下详细的操作步骤。

通过 MDC 实现日志分离

第一步，创建测试接口。

在 Controller 中添加下面的代码，请求资源路径为 /mdc/1 。其中，我们通过 {id} 来判断要不要把 MDC 信息写入日志中。

复制代码

```
    @GetMapping("/mdc/{id}")
    @ResponseBody
    public Object MdcGet(@PathVariable Integer id) {
        HashMap<String, Object> response = new HashMap<>();
        response.put("mdc", "调试");
        response.put("time", new Date());
         if (id == 1) {
            MDC.put("dunshan", "flag");
            log.info("日志写入 " + id);
            MDC.remove("dunshan");
        } else {
            log.info("不是 mdc 日志 " + id);
        }
        return response;
    }
```

第二步，修改配置文件。

代码写好后，需要在 logback-spring.xml 中添加下面的配置：

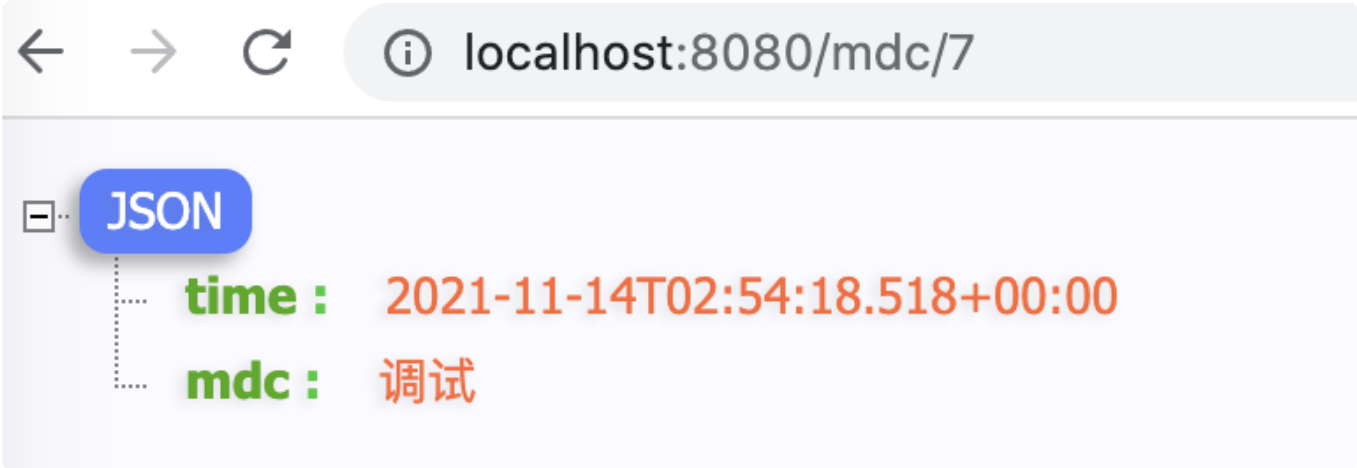
复制代码

```
<pattern>[dunshan :%X{dunshan}]  %d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5lev
```

这个配置主要目的是格式化日志。

第三步，接口测试验证。

在浏览器中发送一次请求，后台日志就会根据规则输出对应内容。



再来看一下服务端日志的显示情况，我们看到 MDC 信息已经成功加入到控制台日志了。

```
[dunshan :]  2021-11-14 10:54:04.712 [http-nio-8080-exec-1] INFO  org.apache.catalina.core.ContainerBase.[Tomcat].[localhost].[/] - Initializing Spring
[dunshan :]  2021-11-14 10:54:04.712 [http-nio-8080-exec-1] INFO  org.springframework.web.servlet.DispatcherServlet - Initializing Servlet 'dispatcherSe
[dunshan :]  2021-11-14 10:54:04.713 [http-nio-8080-exec-1] INFO  org.springframework.web.servlet.DispatcherServlet - Completed initialization in 1 ms
[dunshan :]  2021-11-14 10:54:04.745 [http-nio-8080-exec-1] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 不是 mdc 日志 13
[dunshan :]  2021-11-14 10:54:05.407 [http-nio-8080-exec-2] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 不是 mdc 日志 13
[dunshan :flag]  2021-11-14 10:54:08.327 [http-nio-8080-exec-3] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 日志写入 1
[dunshan :flag]  2021-11-14 10:54:10.603 [http-nio-8080-exec-4] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 日志写入 1
[dunshan :flag]  2021-11-14 10:54:11.162 [http-nio-8080-exec-5] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 日志写入 1
[dunshan :]  2021-11-14 10:54:15.950 [http-nio-8080-exec-6] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 不是 mdc 日志 7
[dunshan :]  2021-11-14 10:54:17.846 [http-nio-8080-exec-7] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 不是 mdc 日志 7
[dunshan :]  2021-11-14 10:54:18.518 [http-nio-8080-exec-8] INFO  com.dunshan.dunshanfile.controller.MdcDomeController - 不是 mdc 日志 7
```

第四步，创建 GenericFilterBean 全局过滤类。

我们通过实现全局 Filter 类 GenericFilterBean 获取标记并加入 MDC 上下文中，这样，后续我们就可以根据 MDC 将压测流量和正常流量的日志分别保存到不同文件目录中去了。

参考代码如下：

复制代码

```
@Component
public class MyFilter extends GenericFilterBean {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servle
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        String flag = request.getHeader("flag");
        if (StringUtils.isNotBlank(flag)) {
            MDC.put("dunshan", "shadow");
            logger.info("flag：" + flag);
        } else {
            MDC.put("dunshan", "produce");
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void destroy() {
        MDC.clear();
    }
}
```

第五步，创建 Logback 配置文件。

标记增加成功后，再在 logback-spring-xml 中添加 Log 输出格式、路径、级别、输出位置以及 MDC 的配置等信息：

复制代码

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- https://github.com/spring-projects/spring-boot/blob/v1.5.13.RELEASE/s
    <include resource="org/springframework/boot/logging/logback/defaults.xml"

    <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg
    <property name="FILE_LOG_PATTERN"
              value="%d{yyyy-MM-dd HH:mm:ss.SSS} ${LOG_LEVEL_PATTERN:-%5p} [%t
    <property name="CONSOLE_LOG_PATTERN"
              value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL

    <property name="APP_NAME" value="dunshan-log" />
    <!--    文件保存路径注意-->
    <property name="LOG_PATH" value="logs/${APP_NAME}/" />
    <property name="LOG_FILE" value="${LOG_PATH}/application.log" />

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <appender name="MDC"
              class="ch.qos.logback.classic.sift.SiftingAppender">
        <discriminator>
            <key>dunshan</key>
            <defaultValue>produce</defaultValue>
        </discriminator>
        <sift>
            <appender name="FILE-${logFileName}" class="ch.qos.logback.core.ro
                <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRol
                    <!-- 按天回滚 daily -->
                    <fileNamePattern>${LOG_PATH}/${dunshan}/log-info-${dunshan
                    <!-- 日志最大的历史 30天 -->
                    <maxHistory>7</maxHistory>
                    <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logb
                        <!-- maxFileSize:这是活动文件的大小，默认值是10MB，这里设置为
                        <maxFileSize>50MB</maxFileSize>
                        <totalSizeCap>20GB</totalSizeCap>
                    </timeBasedFileNamingAndTriggeringPolicy>
                </rollingPolicy>
                <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEn
                    <pattern>${FILE_LOG_PATTERN}</pattern>
                </encoder>
                <filter class="ch.qos.logback.classic.filter.LevelFilter">
```

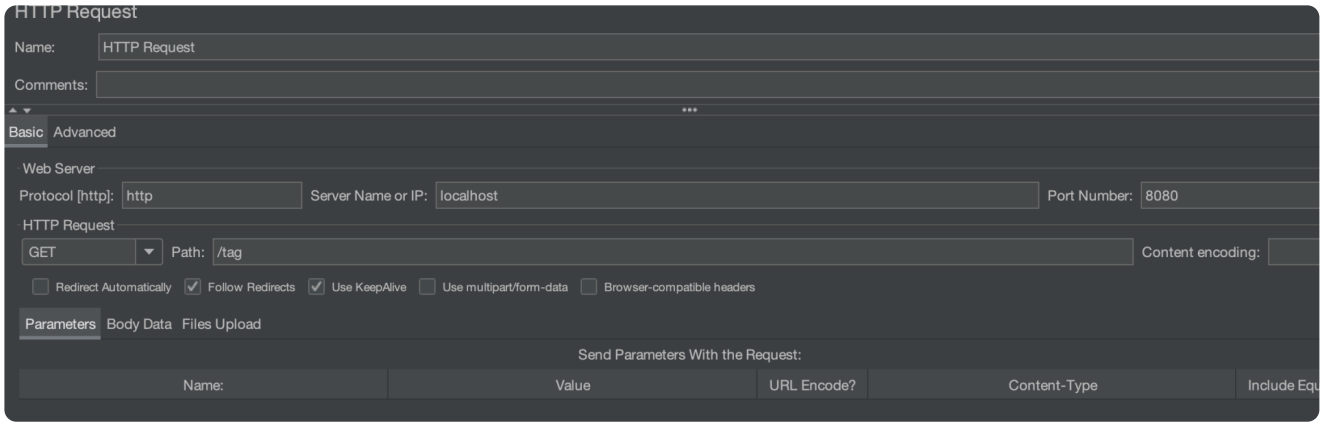
```
                    <!-- 只打印info日志 -->
                    <level>INFO</level>
                    <onMatch>ACCEPT</onMatch>
                    <onMismatch>DENY</onMismatch>
                </filter>
            </appender>
        </sift>
    </appender>


    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="MDC" />
    </root>
</configuration>
```

第六步，接口测试验证。

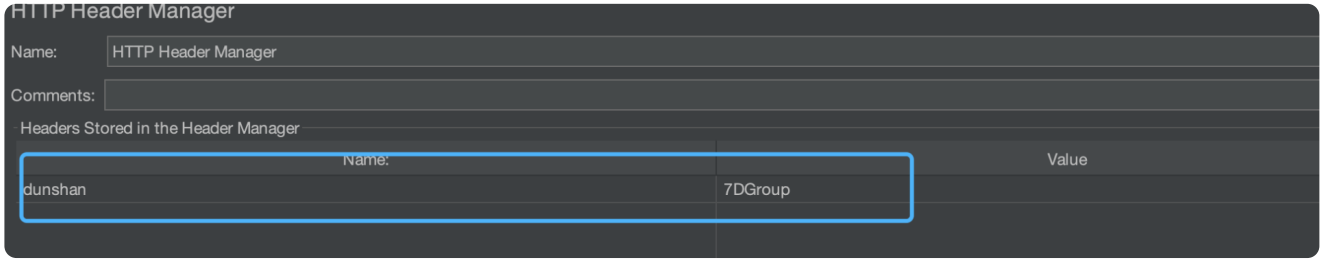
代码添加成功之后，我们还是使用 JMeter 模拟正常流量和压测流量的请求验证一下。

首先，我们使用 JMeter 创建两个请求。

正常流量（不带 Header 标记）



压测流量（带 Header 标记）

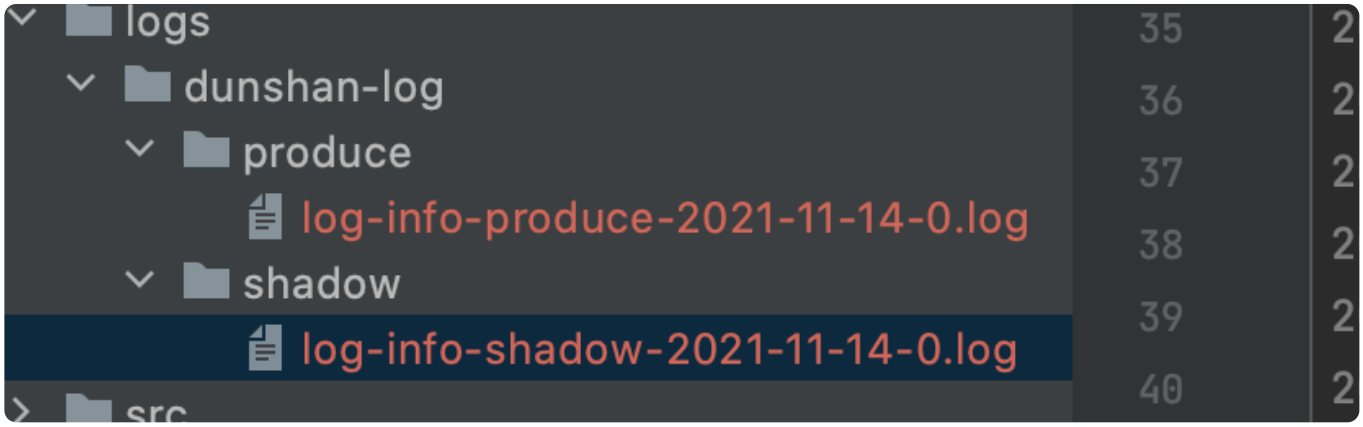


场景设置并发 100 次，循环 100 次，执行后，查看控制台。

```
2021-11-14 12:00:41.053  INFO 25776 --- [nio-8080-exec-8] c.d.d.controller.LogController           : 调试。。
2021-11-14 12:00:41.053  INFO 25776 --- [nio-8080-exec-9] com.dunshan.dunshanfile.config.MyFilter  : flag: 7DGourp
2021-11-14 12:00:41.053  INFO 25776 --- [nio-8080-exec-9] c.d.d.controller.LogController           : 压测。。
2021-11-14 12:00:41.054  INFO 25776 --- [io-8080-exec-10] c.d.d.controller.LogController           : 调试。。
2021-11-14 12:00:41.054  INFO 25776 --- [nio-8080-exec-1] com.dunshan.dunshanfile.config.MyFilter  : flag: 7DGourp
2021-11-14 12:00:41.054  INFO 25776 --- [nio-8080-exec-1] c.d.d.controller.LogController           : 压测。。
2021-11-14 12:00:41.054  INFO 25776 --- [nio-8080-exec-2] c.d.d.controller.LogController           : 调试。。
2021-11-14 12:00:41.055  INFO 25776 --- [nio-8080-exec-3] com.dunshan.dunshanfile.config.MyFilter  : flag: 7DGourp
2021-11-14 12:00:41.055  INFO 25776 --- [nio-8080-exec-3] c.d.d.controller.LogController           : 压测。。
2021-11-14 12:00:41.055  INFO 25776 --- [nio-8080-exec-4] c.d.d.controller.LogController           : 调试。。
2021-11-14 12:00:41.055  INFO 25776 --- [nio-8080-exec-5] com.dunshan.dunshanfile.config.MyFilter  : flag: 7DGourp
2021-11-14 12:00:41.055  INFO 25776 --- [nio-8080-exec-5] c.d.d.controller.LogController           : 压测。。
```

在控制台中，我们可以看到正式流量和压测流量的日志已经被分离。

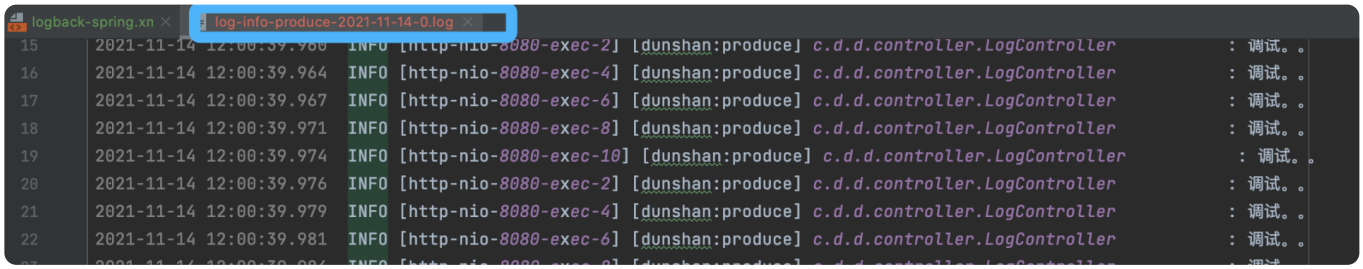
打开配置的日志文件路径，查看日志分离情况。



我们再打开日志看看是否显示正常。

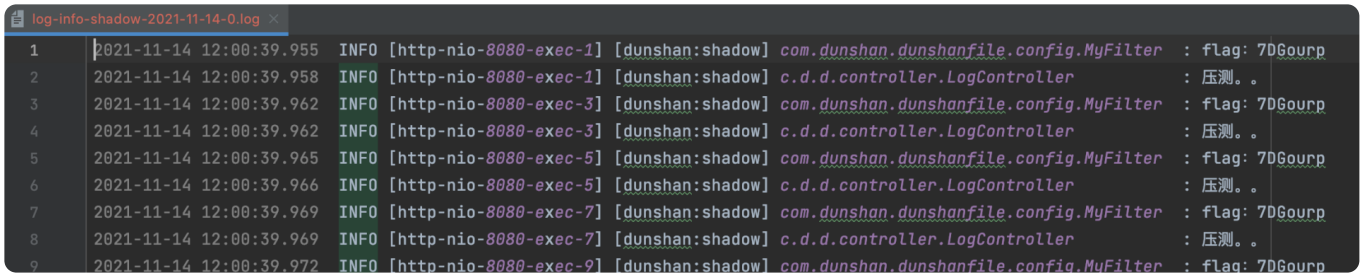
正常日志

先打开正常日志文件，可以看到日志生产正常。


logback-spring.xml
log-info-produce-2021-11-14-0.log
2021-11-14 12:00:39.964 INFO [http-nio-8080-exec-4] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.967 INFO [http-nio-8080-exec-6] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.971 INFO [http-nio-8080-exec-8] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.974 INFO [http-nio-8080-exec-10] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.976 INFO [http-nio-8080-exec-2] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.979 INFO [http-nio-8080-exec-4] [dunshan:produce] c.d.d.controller.LogController : 调试。。
2021-11-14 12:00:39.981 INFO [http-nio-8080-exec-6] [dunshan:produce] c.d.d.controller.LogController : 调试。。

压测日志

再打开影子目录下的压测日志，查看压测日志也生成正常。



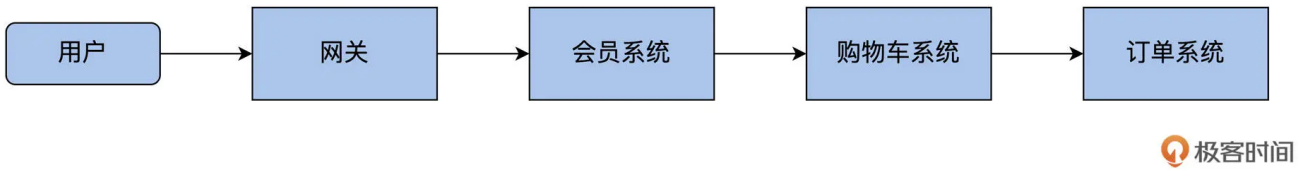
这说明单个服务的日志分离成功了。

刚才我们介绍了日志隔离方案并分别进行了技术预演。预演成功，接下来，我们就要将相关代码移植到 demo 系统进行验证了。

demo 改造验证

首先，我们写个 demo 尝试做一下技术改造验证。

这里使用的 demo 工程还是 14 讲的示例，主要包括网关、会员系统、购物车系统和订单系统 4 个服务：



极客时间

接下来我们主要改造会员系统，实现日志隔离技术预演。

第一步，移植获取标记类。

快速移植全局 Filter 过滤器和数据上下文类：

AppContext；

ContextFilter。

这个操作前面说过很多次了，这里就不再多赘述了。

不过在日志隔离改造环节，网关服务需要额外通过实现全局 Filter 类获取标记，然后加入 MDC 上下文中。

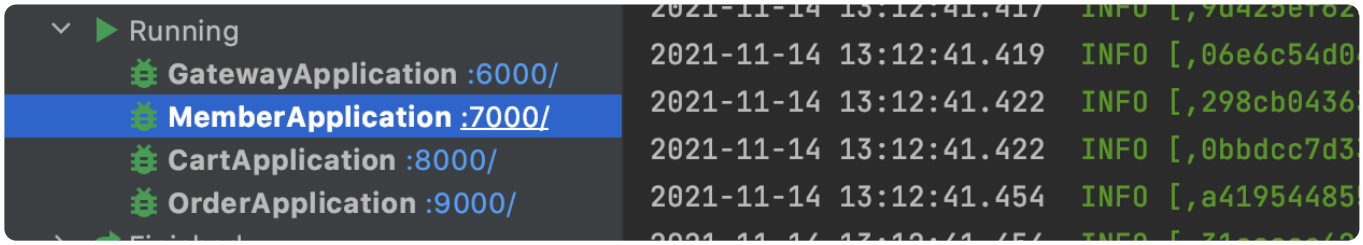
复制代码

```
@Log4j2
@Component
public class ContextFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servle
        BaggageField dunshan = BaggageField.getByName("dunshan");
        String value = dunshan.getValue();
        AppContext appContext = new AppContext();
        if (StringUtils.isNotBlank(value)) {
            log.info("压测流量: " + value);
            appContext.setFlag(value);
            AppContext.setContext(appContext);
            MDC.put("dunshan","shadow");
        } else {
            log.info("正常流量: " + value);
            MDC.put("dunshan","produce");
            AppContext.setContext(appContext);
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void destroy() {
        AppContext.removeContext();
        Filter.super.destroy();

    }
}
```

第二步，移植 Logback 配置文件。

这里我们把上面 Logback-spring-xml 配置文件快速移植到 member 服务，其他服务也可以按上面的配置做相应改造。为了演示的完整性，我们又继续改造了 demo 中的 cart 和 order 服务。

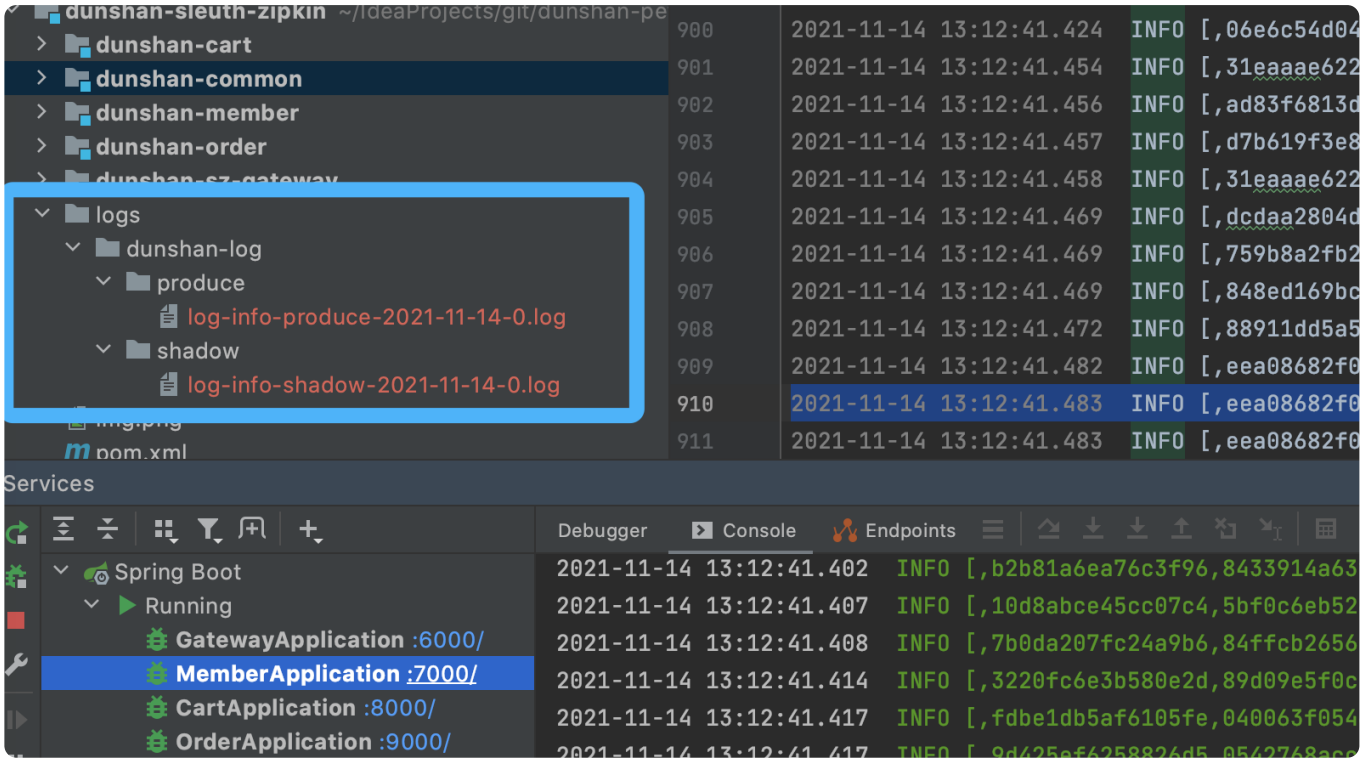
改造完成后，启动网关、memeber、cart 、order 服务：



第三步，接口测试验证。

打开 JMeter 添加请求接口，注意，这个请求也要区分压测流量（带标记的请求）和正常流量（不带标记的请求）。

请求完成后，日志目录显示的是下面的样子。



我们看到，正常日志与压测日志目录是分开的。

但是我们还发现，日志打印是混乱的，日志当中有压测日志，也有正常日志，如下所示：

```
,true] [http-nio-7000-exec-8] [dunshan:produce] com.dunshan.member.config.ContextFilter  : 压测流量: 7DGroup
,true] [http-nio-9000-exec-3] [dunshan:produce] c.d.order.controller.OrderController     : 我是正常流量。。。。。
,true] [http-nio-7000-exec-4] [dunshan:produce] com.dunshan.member.config.ContextFilter  : 正常流量: null
,true] [http-nio-8000-exec-3] [dunshan:produce] com.dunshan.cart.config.ContextFilter    : 压测流量: 7DGroup
,true] [http-nio-9000-exec-14] [dunshan:produce] com.dunshan.order.config.ContextFilter  : 压测流量: 7DGroup
,true] [http-nio-9000-exec-9] [dunshan:produce] c.d.order.controller.OrderController     : 我是正常流量。。。。。
,true] [http-nio-7000-exec-6] [dunshan:produce] com.dunshan.member.config.ContextFilter  : 压测流量: 7DGroup
,true] [http-nio-7000-exec-4] [dunshan:produce] com.dunshan.member.config.ContextFilter  : 压测流量: 7DGroup
,true] [http-nio-9000-exec-6] [dunshan:produce] com.dunshan.order.config.ContextFilter   : 压测流量: 7DGroup
,true] [http-nio-9000-exec-15] [dunshan:produce] c.d.order.controller.OrderController    : 我是正常流量。。。。。
,true] [http-nio-8000-exec-20] [dunshan:produce] com.dunshan.cart.config.ContextFilter   : 正常流量: null
,true] [http-nio-9000-exec-2] [dunshan:produce] com.dunshan.order.config.ContextFilter   : 正常流量: null
,true] [http-nio-9000-exec-2] [dunshan:produce] c.d.order.controller.OrderController     : 我是正常流量。。。。。
```

也就是说，虽然流量已经识别了，日志也分离了，但日志输出是无序的，这显然不是我们想要的结果。

我们知道 MDC 底层使用的是 ThreadLocal，天然就存在这个缺陷。那么，我们这么办呢？所以在这里，我们考虑引入**数据上下文**对象来解决这个问题。

第四步，改造数据上下文类。

修改数据上下文类，我们可以直接把标记存储到 MDC 中去，你可以参考下面的代码：

复制代码

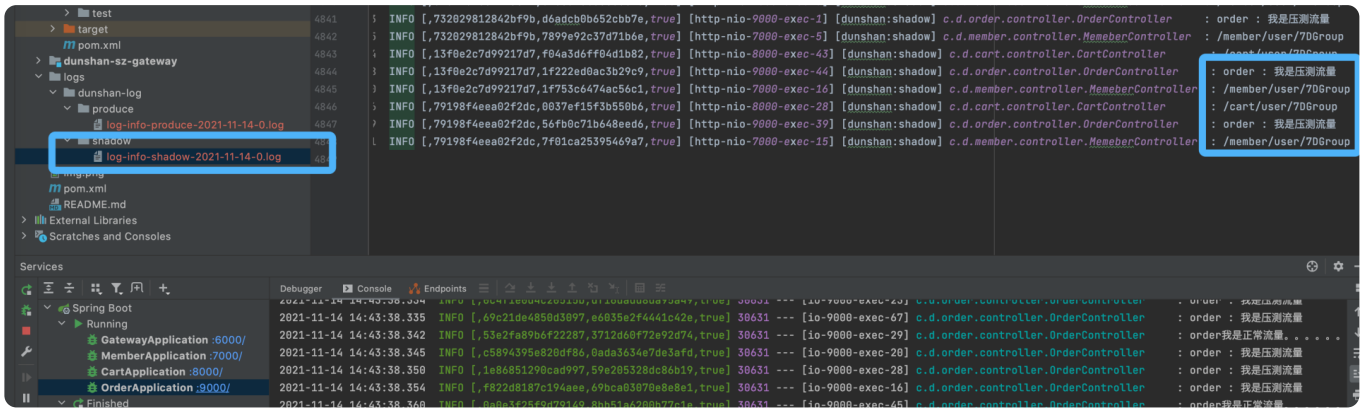
```
@Log4j2
public class AppContext implements Serializable {

    private static final TransmittableThreadLocal<AppContext> contextdunshan =

    private String Flag;

    public static AppContext getContext() {
        return contextdunshan.get();
    }

    public static void setContext(AppContext context) {
        //判断标记
        if (StringUtils.isNotEmpty(context.getFlag())) {
            MDC.put("dunshan", "shadow");
        } else {
            MDC.put("dunshan", "produce");
        }
        contextdunshan.set(context);
    }

    public static void removeContext() {
        contextdunshan.remove();
```

```
    }

    public String getFlag() {
        return Flag;
    }

    public void setFlag(String Flag) {
        this.Flag = Flag;
    }

}
```

第五步，JMeter 接口测试验证。

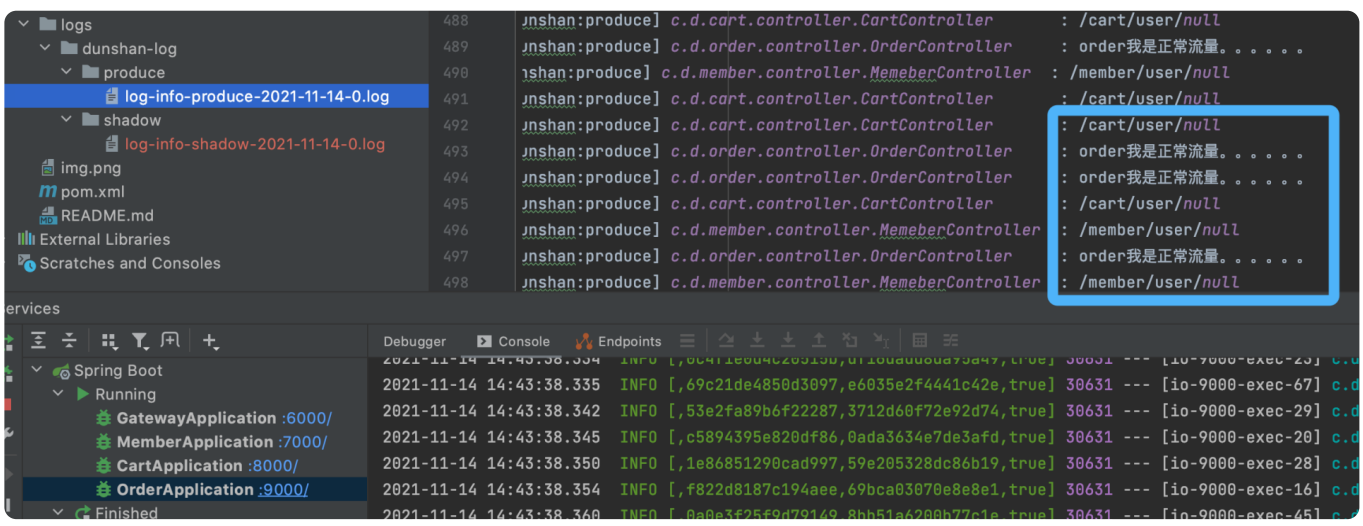
我们再次启动项目，执行第三步的 JMeter 脚本发送请求。

执行完毕后，我们打开影子目录中的最新日志文件查看一下内容。



可以看到，保存的全部是压测日志，这才是合理的。

我们再打开正常目录中的日志文件查看下内容。



可以看到，保存的全部都是正常日志。很好，这就是我们想要的日志隔离效果。

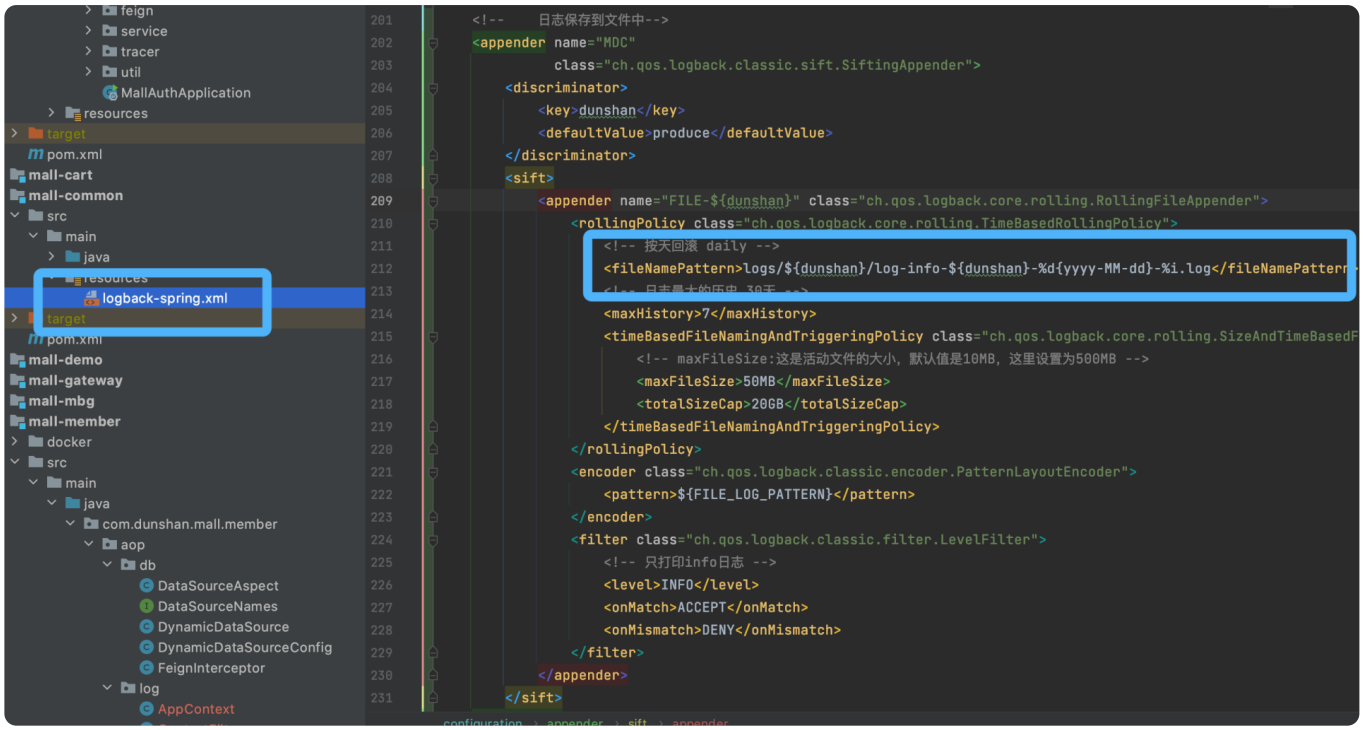
经过刚才的改造，我们 member、cart、order 服务的日志都已经隔离成功了，下面，我们就尝试将这个 demo 移植到真实系统中去进行改造了。

真实系统改造

打开 mall 项目，我们还是先对 member 系统进行改造，改造成功后，再移植到其他服务中去。

第一步，修改 Logback 配置文件。

把之前 demo 的配置文件放到 mall 中的 logback-spring.xml 中：



第二步，修改网关全局过滤器 AuthGlobalFilter。

在 AuthGlobalFilter 类中加入 MDC 方法对流量标记进行判断 ，具体判断代码是 stringutils.isNotEnpty(dunshan)，如下图所示：

```
try {
    if (StringUtils.isNotEmpty(dunshan)) {
        MDC.put( key: "dunshan",  val: "shadow");
        LOGGER.info("dunshan-流量标识: " + dunshan);
        tracer.currentSpan().tag(AuthConstant.TRAFFIC_SIGNS, dunshan);
    } else {
        MDC.put( key: "dunshan",  val: "produce");
    }
    //从token中解析用户信息并设置到Header中去
    String realToken = token.replace(AuthConstant.JWT_TOKEN_PREFIX,  replacement: "");
    JWSObject jwsObject = JWSObject.parse(realToken);
    String userStr = jwsObject.getPayload().toString();
    LOGGER.info("AuthGlobalFilter.filter() user:{}", userStr);
    ServerHttpRequest request = exchange.getRequest().mutate().header(AuthConstant.USER_TOKEN_
    exchange = exchange.mutate().request(request).build();
```

第三步，修改业务系统数据上下文类。

接下来，我们改造对应业务系统 mall-member 、mall-auth 项目的数据上下文类，可以直接把标记存储到 MDC 中。

参考代码如下：

```
FeignInterceptor
log
  AppContext
  ContextFilter
mongodb
  MongoSwitch
  MultiMongoTemplate
redb
  RedisAspectAop.java
component
  FeignRequestInterceptor
config
  FeignConfigDome
  MyBatisConfig
  RedisConfig
  SwaggerConfig
controller
  MemberAttentionControll
  MemberController
  MemberCouponControlle
  MemberProductCollectio
  MemberReadHistoryCont
  MemberReceiveAddress(
```

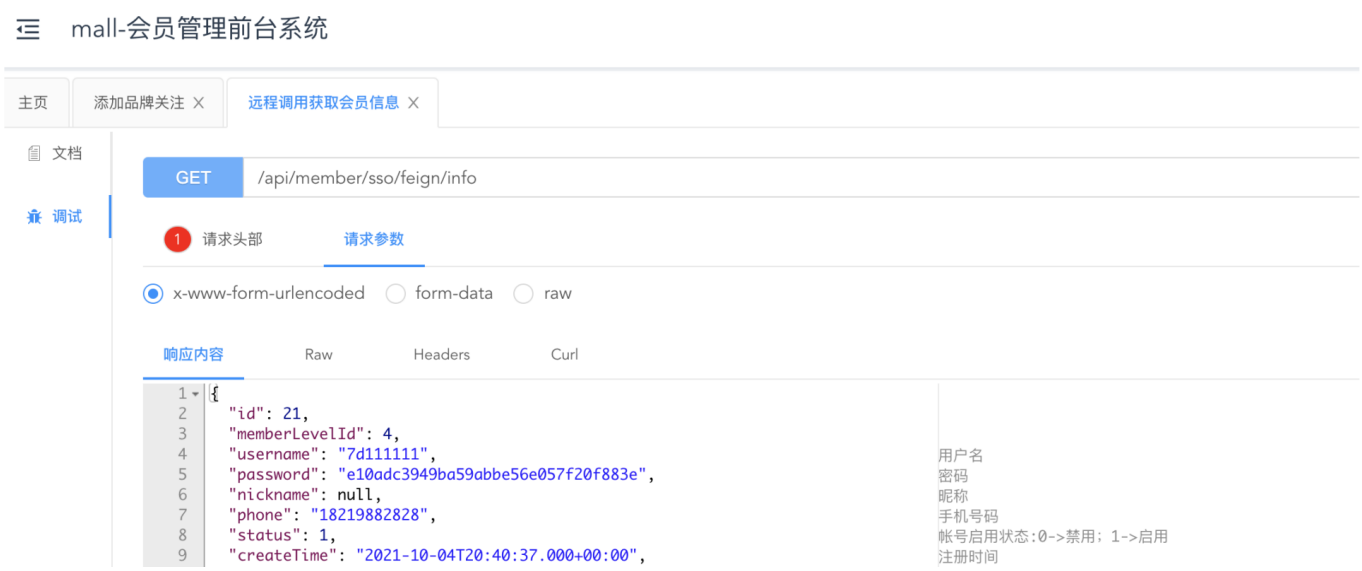
```
16
17      private static final TransmittableThreadLocal<AppContext> contextdunshan = new TransmittableThreadLocal<>();
18
19      private String traceId;
20
21      public static AppContext getContext() { return contextdunshan.get(); }
24
25  @   public static void setContext(AppContext context) {
26          if (StringUtils.isNotEmpty(context.getTraceId())) {
27              MDC.put( key: "dunshan",  val: "shadow");
28          } else {
29              MDC.put( key: "dunshan",  val: "produce");
30          }
31          contextdunshan.set(context);
32      }
33
34      public static void removeContext() { contextdunshan.remove(); }
37
38      public String getTraceId() { return traceId; }
41
42      public void setTraceId(String traceId) { this.traceId = traceId; }
45  }
```

第四步，接口测试验证。

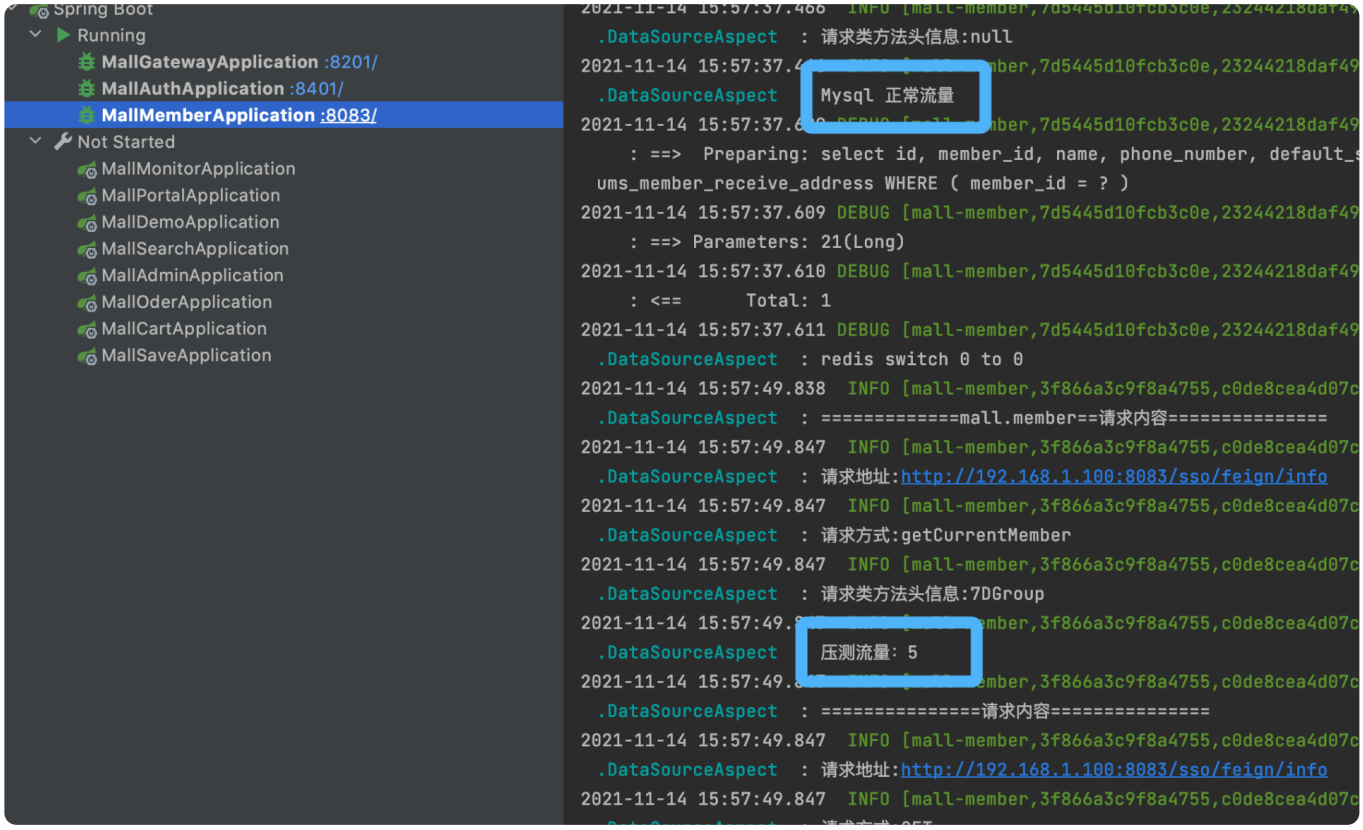
我们启动网关服务、认证服务和会员服务来验证一下改造结果。

正常流量（不带 Header 标记）

首先，我们模拟发送一下正式流量。打开接口文档，测试下会员服务的【获取用户信息接口】。



我们可以在控制台看到正常的日志。



接下来，我们查看下正常的日志文件内容。

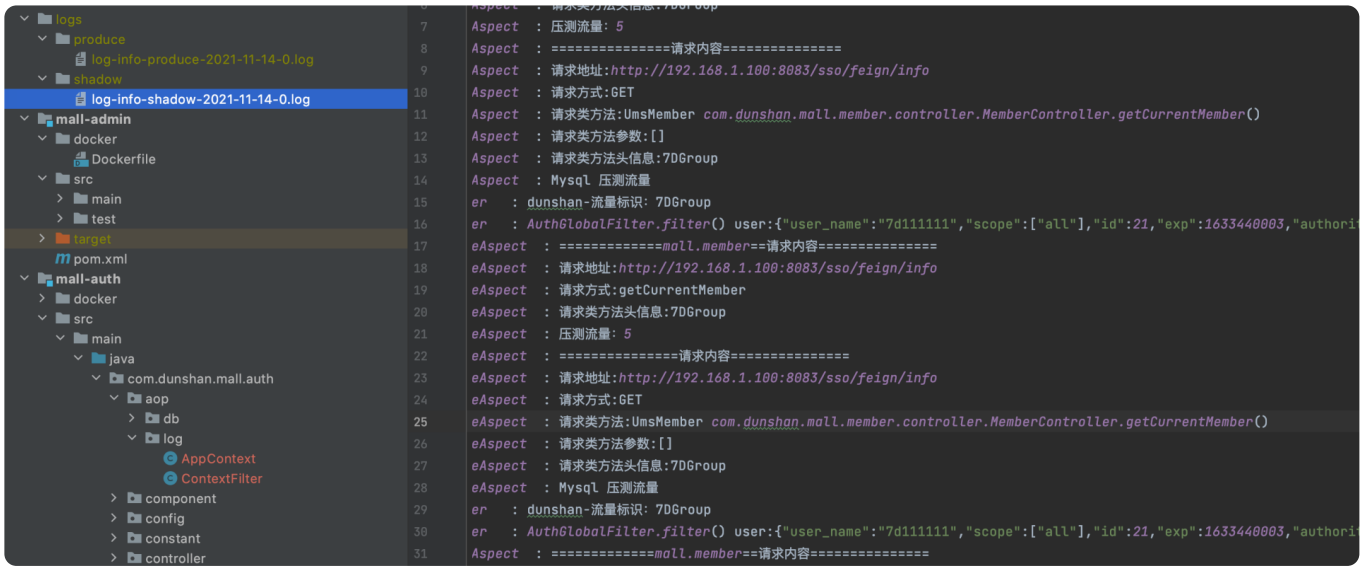
可以看到，正常日志生成成功。

压测流量（带 Header 标记）

接下来，我们再模拟发送一下压测请求。我们通过 curl 命令增加 Header 标记 ，然后发送一个请求，模拟压测流量。具体命令可以参考下面的代码：

```
curl -X GET -H "Accept:*/*" -H "Authorization:eyJhbGciOiJSUzI1NiIsInR5cCI6Ik
```

测试完成后，我们找到影子目录下的压测日志文件：



可以看到，压测日志也生成成功了。

到这里，我们的日志隔离改造就成功了。

另外，我还想说明一点，你可能已经发现了，正常请求和压测请求的用户是一样的。这一点其实我们在《铺底数据：真实的压测数据应该做成什么样子？》这篇文章中就讲过，因为压测用户是从线上数据脱敏来的，所以正常用户与压测用户是一致的。

有了上面成功的改造经验，其他的服务就也可以按照目前的方式改造了。

总结

好了，这节课就讲到这里。刚才，我们完整演示了日志隔离的技术预演和真实系统改造过程。

日志系统的混乱是每个定位问题人员的恶梦，我想你可能也经历过在微服务分布式架构的汪洋一样的日志中查找问题的痛苦经历。所以日志聚合系统是微服务分布式架构的必备基础设施。

而在全链路压测的逻辑中，如果没有把压测流量和正式流量做隔离，那出现问题时，查找问题就会累到跳脚了；做日志隔离还有一个好处，就是在日志隔离之后，我们可以把压测日志输出到单独的存储上去，这样就不会因为压测日志增长过快而导致存储空间很快被耗光了。

在我看来，日志隔离是全链路压测过程中非常必要的改造步骤。刚才，我们演示了“MDC+Logback+ 数据上下文”的日志隔离实现逻辑，这是我认为最为合理的解决方案。

课后题

学完这节课，我想请你思考两个问题：

1. 除了本文的日志隔离方案，你还有其他的日志隔离方案吗？都有什么优缺点？
2. 你知道有哪些海量日志抽取的技术方案？

欢迎你在留言区与我交流讨论。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 流量隔离：RabbitMQ 消息隔离是怎么做的？

下一篇 20 | Mock：如何屏蔽第三方接口的影响？

更多学习推荐

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。